Chapter 1

Introduction: Strings and Trees

1.1 Computational Linguistics: Course Overview

In this class, we will study:

1. Automata Theory
2. Haskell
3. . . . as they pertain to problems in linguistics:
   (a) *Well-formedness* of linguistic representations
   (b) *Transformations* from one representation to another

1.1.1 Linguistic Theory

Linguistic theory often distinguishes between *well- and ill-formed representations.*

Strings. In English, we can coin new words like *bling.* What about the following?

1. *gding*
2. *θwɪk*
3. *spɪf*

Trees. In English, we interpret the compound *deer-resistant* as an adjective, not a noun. What about the following?

1. *green-house*
2. *dry-clean*
3. *over-throw*

Linguistic theory is often also concerned with *transformations.*

Strings. In generative phonology, underlying representations of words are *transformed* to surface representations of words.

1. /kæt-z/ → [kæts]
2. /wɪʃ-z/ → [wɪʃɪz]

Trees. In derivational theories of generative syntax, the deep sentence structure is *transformed* into a surface structure.

1. Mary won the competition.
   (a) The competition was won by Mary.
   (b) What did Mary win?

1.1.2 Automata Theory

Automata are *abstract* machines that answer questions like these.

The Membership Problem

Given: A possibly infinite set of strings (or trees) X.

Input: A input string (or tree) x.

Problem: Does x belong to X?

The Transformation Problem

Given: A possible infinite function of strings to strings (or trees to trees) $f : X \to Y$.

Input: A input string (or tree) x.

Problem: What is $f(x)$?

There are many kinds of automata. Two common types of automata address these specific problems.

Recognizers Recognizers solve the membership problem.
Transducers Transducers solve the transformation problem.

Different kinds of automata instantiate different kinds of memory.

Finite-state Automata An automata is finite-state whenever the amount of memory necessary to solve a problem for input x is fixed and **independent** of the size of x.

Linear-bounded Automata An automata is linear-bounded whenever the amount of memory necessary to solve a problem for input x is **bounded by a linear function** of the size of x.

In this class we will study finite-state recognizers and transducers. There are many types of these as well, some are shown below.

- deterministic vs. non-deterministic

- 1way vs. 2way (for strings)
- bottom-up vs. top-down vs. walking (for trees)

The simplest type is the deterministic, 1way recognizer for strings. We will start there and then complicate them bit by bit:

1. add non-determinism
2. add output (transducers)
3. add 2way-ness
4. generalize strings to trees and repeat

What do automata mean for linguistic theory?

Fact 1: Finite-state automata over strings are sufficient for phonology (Johnson, 1972; Kaplan and Kay, 1994).

Fact 2: Finite-state automata over strings are *NOT* sufficient for syntax, but linear-bounded automata are (Chomsky, 1956; Shieber, 1985, among others).

Fact 3: Finite-state automata over trees *ARE* sufficient for syntax (Rogers, 1998; Kobele, 2011; Graf, 2011).

Hypothesis: Linguistic phenomena can be modeled with special kinds of finite-state automata with even stricter memory requirements over the right representations (Graf, 2015; Heinz, 2018).

Chapter 2

Strings and Trees

In this section, we define strings and trees as recursive data structures of finite size.

2.0.1 Strings

Informally, strings are sequences of symbols.

What are symbols? It is standard to assume a set of symbols called the *alphabet*. The Greek symbol Σ is often used to represent the alphabet but people also use S, A, or anything else. The symbols can be anything: IPA letters, morphemes, words, part-of-speech categories. Σ can be infinite in size, but we will usually consider it to be finite.

Formally, strings are defined inductively with a constructive operation. We will write this operation with "$\cdot$". This symbol typically denotes *concatenation*, which is an operation like addition. Informally, concatenation of strings x and y returens the string xy. We will eventually define concatenation formally. So for now, it is more appropriate to think of "$\cdot$" as a *constructor* which tells one how to build new structures from existing ones. Before we can define operations over strings like concatenation, we need to know *what strings are*.

Inductive definitions are defined in two parts: the *base case* and an *inductive case*.

Definition 1 (Strings, version 1)**.**

Base Case: *If $a \in \Sigma$ then a is a string.*
Inductive Case: *If $a \in \Sigma$ and w is a string then $a \cdot (w)$ is a string.*

Example 1. Let $\Sigma = \{a, b, c\}$. Then the following are strings.

1. $a \cdot (b \cdot (c))$
2. $a \cdot (a \cdot (a))$
3. $a \cdot (b \cdot (c \cdot (c)))$

Frankly, writing all the parentheses and "$\cdot$" is cumbersome. So the above examples are much more readable if written as follows.

1. abc

2. *aaa*
3. *abcc*

Writing all the parentheses and "·" is also unnecessary because the above definition provides a unique "derivation" for each string.

Example 2. Let $\Sigma = \{a, b, c\}$. We claim $w = a \cdot (b \cdot (c \cdot (c)))$ is a string. There is basically one way to show this. First we observe that $a \in \Sigma$ so whether w is a string depends on whether $x = b \cdot (c \cdot (c))$ is a string by the inductive case. Next we observe that since $b \in \Sigma$ whether x is a string depends on whether $y = c \cdot (c)$ is a string, again by the inductive case. Once more, since $c \in \Sigma$ whether y is a string depends on whether c is a string. Finally, by the base case c is a string and so the dominoes fall: y is a string so x is a string and so w is a string.

This unique derivabality is useful in many ways. For instance, suppose we want to determine the length of a string. Here is how we can do it.

Definition 2 (string length)**.** *The* length *of a string w, written $|w|$, is defined as follows. If there is $a \in \Sigma$ such that $w = a$ then $|w| = 1$. If not, then $w = a \cdot (x)$ where x is some string and $a \in \Sigma$. In this case, $|w| = |x| + 1$.*

Note length is an inductive definition!

Example 3. What is the length of string $w = abcc$? Well, as before we see that $w = a \cdot x$ where $x = bcc$. Thus, $|w| = 1 + |bcc|$. What is the length of bcc? Well, $x = b \cdot y$ where $y = cc$. So now we have $|w| = 1 + (1 + |cc|)$. Since $y = c \cdot z$ where $z = c$ we have $|w| = 1 + (1 + (1 + |c|))$. Finally, by the *base case* we have $|w| = 1 + (1 + (1 + (1))) = 4$.

There is another way to define strings, which makes use of the so-called empty string. The empty string is usually written with one of the Greek letters ϵ or λ. It's just a matter of personal preference. The empty string is useful from a mathematical perspective in the same way the number zero is useful. Zero is a special number because for all numbers x it is the case that $0 + x = x + 0 = x$. The empty string serves the same special purpose. It is the unique string with the following special property with respect to concatenation.[1]

$$\text{For all strings } w,\ \lambda \cdot w = w \cdot \lambda = w \tag{2.1}$$

With this concept, we can redefine strings as the following.

Definition 3 (Strings, version 2)**.**

Base Case: *λ is a string.*

[1]Technically, concatenation is not yet defined between strings since it is only defined for a string and a symbol. Our goal would be to ensure the property mentioned holds once concatenation is defined between strings.

Inductive Case: *If $a \in \Sigma$ and w is a string then $a \cdot (w)$ is a string.*

As a technical matter, when we write the string abc, we literally mean the following structure: $a \cdot (b \cdot (c \cdot (\lambda)))$.

The definition for length can be redefined similarly.

We can now define concatenation between strings. First we define ReverseAppend, which takes two strings as arguments and returns a third string.

Definition 4 (reverse append)**.** Reverse append *is a binary operation over strings, which we denote $\otimes_{\texttt{revapp}}$. You can also think of it as a function which takes two strings w_1 and w_2 as arguments and returns another string. Here is the base case. If $w_1 = \lambda$ then it returns w_2. So we can write $\lambda \otimes_{\texttt{revapp}} w = w$. Otherwise, there is $a \in \Sigma$ such that $w_1 = a \cdot (x)$ for some string x. In this case,* reverse append *returns $x \otimes_{\texttt{revapp}} a \cdot (w_2)$.*

Exercise 1. Work out what $abc \otimes_{\texttt{revapp}} def$ equals.

Exercise 2. What is $abc \otimes_{\texttt{revapp}} \lambda$? Write a definition for string reversal.

Exercise 3. Define the concatenation of two strings w_1 and w_2 using reverse append and string reversal. Prove this definition satisfies Equation 2.1.

The set of all strings of finite length from some alphabet Σ, including the empty string, is written Σ^*. A *stringset* is a subset of Σ^*.

Stringsets are often called *formal languages.* From a linguistic perspective, is is the study of string well-formedness.

2.0.2 Trees

Trees are like strings in that they are recursive structures. Informally, trees are structures with a single 'root' node which dominates a sequence of trees.

Formally, trees extend the dimensionality of string structures from 1 to 2. In addition to linear order, the new dimension is dominance.

Unlike strings, we will not posit "empty" trees because every tree has a root.

Like strings, we assume a set of symbols Σ. This is sometimes partitioned into symbols of different types depending on whether the symbols can only occur at the leaves of the trees or whether they can dominate other trees. We don't make such a distinction here.

Definition 5 (Trees)**.** *If $a \in \Sigma$ and w is a string of trees then $a[w]$ is a tree.*

If $w = t_1 \cdot (t_2 \cdot (\ldots \cdot (t_n \cdot (\lambda)) \ldots))$, we write $a[t_1 t_2 \ldots t_n]$ for readability. This means the node a dominates trees $t_1, t_2 \ldots t_n$ and that trees $t_1, t_2 \ldots t_n$ are ordered sequentially. A tree $a[\lambda]$ is called a *leaf* or *leaf node.* Note if $w = \lambda$ we typically write $a[\ ]$ instead of $a[\lambda]$.

The definition of trees above may appear circular. It appears circular since it defines trees in terms of strings of trees. However, this circularity is an illusion. The definition has a solid recursive base case, as I will now explain. The key to resolving this illusion is to construct the full set of trees in steps. For example, λ is a string (of trees) by the definition of string. With the empty string λ and the finite alphabet Σ we can define a set of trees $\mathrm{T}_0 = \{a[\lambda] \mid a \in \Sigma\}$. T_0 is the set of all logically possible leaves (trees of depth 0). T_0 is a finite alphabet and so T_0^* is a well defined set of strings over this alphabet. For example $w = a[\,] \cdot (b[\,] \cdot (c[\,] \cdot (b[\,] \cdot (\lambda))))$ is a string of trees.

So far, with Σ and λ we built T_0. The definition of strings gives us T_0^*. Now with Σ and T_0^* we can build $\mathrm{T}_1 = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_0^*\} \cup \mathrm{T}_0$. T_1 includes T_0 in addition to all trees of depth 1. With T_1, and the definition of strings we have T_1^*. Now with Σ and T_1^* we can build $\mathrm{T}_2 = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_1^*\} \cup \mathrm{T}_1$.

More generally, we define $\mathrm{T}_{n+1} = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_n^*\} \cup \mathrm{T}_n$. Finally, let the set of all logically possible trees be denoted with $\Sigma^{\mathrm{T}} = \bigcup_{i \in \mathbb{N}} \mathrm{T}_i$. Figure 2.1 illustrates this construction.

Figure 2.1: The inductive definition of trees.

$$\begin{tikzcd}
 & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \Sigma \arrow[d] & & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \\
\lambda \arrow[r] & \mathrm{T}_0 \arrow[r] & \mathrm{T}_0^* \arrow[r] & \mathrm{T}_1 \arrow[r] & \mathrm{T}_1^* \arrow[r] & \mathrm{T}_2 \arrow[r] & \cdots \arrow[r] & \mathrm{T}_n \arrow[r] & \mathrm{T}_n^* \arrow[r] & \mathrm{T}_{n+1}
\end{tikzcd}$$

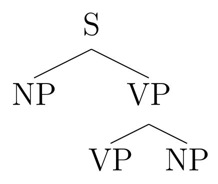
Here are some examples of trees.

Example 4. Let $\Sigma = \{$NP, VP, S$\}$. Then the following are trees.

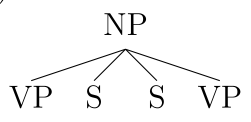
1. S[NP[ ] VP[VP[ ] NP[ ]]]
2. NP[VP[ ] S[ ] S[ ] VP[ ]]
3. NP[NP[NP[ ] VP[ ] S[ ]]]

We might draw these structures as follows.

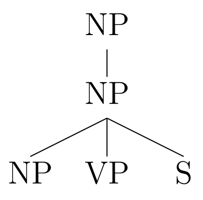
(1)



(2)



(3)



Regarding the tree in (1), its leaves are NP, VP, and NP.

As before, we can now write definitions to get information about trees. For instance here is a definition which gives us the number of nodes in the tree.

Definition 6. *The* size *of a tree t, written $|t|$, is defined as follows. If there is some $a \in \Sigma$ such that $t = a[\,]$ then its size is* 1*. If not, then $t = a[t_1 t_2 \ldots t_n]$ where $a \in \Sigma$ and each t_i is a tree. Then $|t| = 1 + |t_1| + |t_2| + \ldots + |t_n|$.*

Exercise 4. Using the above definition, calculate the size of the trees (1)-(3) above. Write out the calculation explicitly.

Here is a definition for the *width* of a tree.

Definition 7. *The* depth *of a tree t, written* $\textbf{\texttt{depth}}(t)$*, is defined as follows. If there is some $a \in \Sigma$ such that $t = a[\ ]$ then its depth is 0. If not, then $t = a[t_1 t_2 \dots t_n]$ where $a \in \Sigma$ and each t_i is a tree. Then* $\textbf{\texttt{depth}}(t) = 1 + \texttt{max}\big\{\texttt{depth}(t_1), \texttt{depth}(t_2), \dots, \texttt{depth}(t_n)\big\}$ *where* `max` *takes the largest number in the set.*

Definition 8. *The* width *of a tree t, written* $\textbf{\texttt{width}}(t)$*, is defined as follows. If there is some $a \in \Sigma$ such that $t = a[\ ]$ then its width is 0. If not, then $t = a[t_1 t_2 \dots t_n]$ where $a \in \Sigma$ and each t_i is a tree. Then* $\textbf{\texttt{width}}(t) = \texttt{max}\big\{n, \texttt{width}(t_1), \texttt{width}(t_2), \dots, \texttt{width}(t_n)\big\}$ *where* `max` *takes the largest number in the set.*

The set of trees Σ^{T} contains all trees of arbitrary width. Much research also effectively concerns the set of all and only those trees whose width is bounded by some number n. Let $\Sigma^{\mathrm{T}(n)} = \{t \in \Sigma^{\mathrm{T}} \mid \texttt{width}(t) \leq n\}$.

Exercise 5. The *yield* of a tree t, written $\texttt{yield}(t)$, maps a tree to a string of its leaves. For example let t be the tree in (1) in Example 4 above. Then its yield is the string "NP VP NP".

2.0.3 String Exercises

Exercise 6. Let Σ be the set of natural numbers. So we are considering strings of numbers.

1. Write the definition of the function *addOne* which adds one to each number in the in the string. So *addOne* would change the string $5 \cdot (11 \cdot (4 \cdot (\lambda)))$ to $6 \cdot (12 \cdot (5 \cdot (\lambda)))$. Using this definition, show *addOne* of the following number strings is calculated.
   (a) $11 \cdot (4 \cdot (\lambda))$
   (b) $3 \cdot (2 \cdot (\lambda))$
   (c) λ
2. Write the definition of the *timesTwo* of the numbers in the string. So *timesTwo* would change the string $5 \cdot (11 \cdot (4 \cdot (\lambda)))$ to $10 \cdot (22 \cdot (8 \cdot (\lambda)))$. Using this definition, show *timesTwo* of the following number strings is calculated.
   (a) $11 \cdot (4 \cdot (\lambda))$
   (b) $3 \cdot (2 \cdot (\lambda))$
   (c) λ

Exercise 7. Let Σ be the set of natural numbers. So we are considering strings of numbers.

1. Write the definition of the *sum* of the numbers in the string. Using this definition, show how the sum of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) λ

2. Write the definition of the *product* of the numbers in the string. Using this definition, show how the sum of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) λ

2.0.4 Tree Exercises

Exercise 8. Let Σ be the set of natural numbers. Now let's consider trees of numbers.

1. Write the definition of the function *addOne* which adds one to each number in the tree. Using this definition, calculate *addOne* as applied to the trees below.

   (a) 4[12[ ] 3[ ]]

   (b) 4[12[ ] 3[1[ ] 2[ ]]]

   (c) 4[12[7[ ] 7[6[ ]]] 3[1[ ] 2[ ]]]

2. Write the definition of the function *isLeaf* which changes the nodes of a tree to `True` if it is a leaf node or to `False` if it is not. Using this definition, calculate *isLeaf* as applied to the trees below.

   (a) 4[12[ ] 3[ ]]

   (b) 4[12[ ] 3[1[ ] 2[ ]]]

   (c) 4[12[7[ ] 7[6[ ]]] 3[1[ ] 2[ ]]]

Exercise 9. Let Σ be the set of natural numbers. Now let's consider trees of numbers.

1. Write the definition of the *sum* of the nodes of the tree. Using this definition, show how the sum of the following number trees is calculated.

   (a) 4[12[ ] 3[ ]]

   (b) 4[12[ ] 3[1[ ] 2[ ]]]

   (c) 4[12[7[ ] 7[6[ ]]] 3[1[ ] 2[ ]]]

2. Write the definition *max* which takes a tree as input and returns the largest valued node in the tree.

   (a) 4[12[ ] 3[ ]]

   (b) 4[12[ ] 3[1[ ] 2[ ]]]

   (c) 4[12[7[ ] 7[6[ ]]] 3[1[ ] 2[ ]]]

Bibliography

Chomsky, Noam. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 113–124. IT-2.

Comon, H., M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree automata techniques and applications. Available on: `http://tata.gforge.inria.fr/`. Release October, 12th 2007.

Graf, Thomas. 2011. Closure properties of Minimalist derivation tree languages. In *LACL 2011*, edited by Sylvain Pogodalla and Jean-Philippe Prost, vol. 6736 of *Lecture Notes in Artificial Intelligence*, 96–111. Heidelberg: Springer.

Graf, Thomas. 2015. Computational unity across language modules. Invited talk, December 15, Department of Theoretical and Applied Linguistics, Moscow State University, Moscow, Russia.

Heinz, Jeffrey. 2018. The computational nature of phonological generalizations. In *Phonological Typology*, edited by Larry Hyman and Frans Plank, Phonetics and Phonology. Mouton. Final version submitted November 2015. Expected publication in 2017.

Heinz, Jeffrey, and Regine Lai. 2013. Vowel harmony and subsequentiality. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, edited by Andras Kornai and Marco Kuhlmann, 52–63. Sofia, Bulgaria.

Hopcroft, John, Rajeev Motwani, and Jeffrey Ullman. 2001. *Introduction to Automata Theory, Languages, and Computation*. Boston, MA: Addison-Wesley.

Johnson, C. Douglas. 1972. *Formal Aspects of Phonological Description*. The Hague: Mouton.

Kaplan, Ronald, and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics* 20:331–378.

Kobele, Gregory M. 2011. Minimalist tree languages are closed under intersection with recognizable tree languages. In *LACL 2011*, edited by Sylvain Pogodalla and Jean-Philippe Prost, vol. 6736 of *Lecture Notes in Artificial Intelligence*, 129–144. Berlin: Springer.

Rogers, James. 1998. *A Descriptive Approach to Language-Theoretic Complexity*. Stanford, CA: CSLI Publications.

Shieber, Stuart. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy* 8:333–343.